Java 18 Features

**Date 11/05/2023**

# Overview of Java 18 (Mar 22, 2022)

1. JEP 400 - UTF-8 by Default
2. JEP 420 - Pattern Matching for Switch
3. JEP 408 - Simple Web Server
4. JEP 418 - Internet-Address Resolution SPI

# JEP 400 - UTF-8 by Default

In Java 18, this JEP makes the default charset to UTF-8. However, we still allow configuring the default charset to others by providing the system property ‘file.encoding’.

In Java 18, if the file.encoding system property is COMPACT, the JVM uses Java 17 and an earlier algorithm to choose the default charset.

**java -Dfile.encoding=COMPAT**

If the file.encoding is UTF-8, the default charset will be UTF-8, a no-op value in Java 18.

**java -Dfile.encoding=UTF-8**

Before Java 18, the default charset was environment-dependent, meaning the Java Virtual Machine (JVM) chooses the default charset during start-up, based on the run-time environment, like the operating system, the user’s locale, and other factors. For example, on macOS, the default charset is UTF-8; on Windows, the default charset is ‘windows-1252’ (if English locale).

Since the default charset is not the same from machine to machine, APIs that use the default charset may cause unwanted behaviors or errors, especially the IO APIs like java.io.FileReader and java.io.FileWriter.

For a long time, we Java developers have had to deal with the fact that the standard Java character set varies depending on the operating system and language settings.

## The Problem

The Java standard character set determines how Strings are converted to bytes and vice versa in numerous methods of the JDK class library (e.g., when writing and reading a text file). These include, for example:

the constructors of FileReader, FileWriter, InputStreamReader, OutputStreamWriter,

the constructors of Formatter and Scanner,

the static methods URLEncoder.encode() and URLDecoder.decode().

This can lead to unpredictable behavior when an application is developed and tested in one environment – and then run in another (where Java chooses a different default character set).

For example, let's run the following code on Linux or macOS (the Japanese text is "Happy Coding!" according to Google Translate):

try (FileWriter fw = new FileWriter("happy-coding.txt");

BufferedWriter bw = new BufferedWriter(fw)) {

bw.write("ハッピーコーディング！");

}

And then, we load this file with the following code on Windows:

try (FileReader fr = new FileReader("happy-coding.txt");

BufferedReader br = new BufferedReader(fr)) {

String line = br.readLine();

System.out.println(line);

}

Then the following is displayed:

**ãƒ?ãƒƒãƒ”ãƒ¼ã‚³ãƒ¼ãƒ‡ã‚£ãƒ³ã‚°ï¼?**

That is because Linux and macOS store the file in UTF-8 format, and **Windows tries to read it in Windows-1252 format**.

## The Problem – Stage Two

It becomes even more chaotic because newer class library methods do not respect the default character set but always use UTF-8 if no character set is specified. These methods include, for example, Files.writeString(), Files.readString(), Files.newBufferedWriter(), and Files.newBufferedReader().

Let's start the following program, which writes the Japanese text via FileWriter and reads it directly afterward via Files.readString():

try (FileWriter fw = new FileWriter("happy-coding.txt");

BufferedWriter bw = new BufferedWriter(fw)) {

bw.write("ハッピーコーディング！");

}

String text = Files.readString(Path.of("happy-coding.txt"));

System.out.println(text);

Linux and macOS display the correct Japanese text. On Windows, however, we see only question marks:

**???????????**

That is because, on Windows, FileWriter writes the file using the standard Java character set Windows-1252, but Files.readString() reads the file back in as UTF-8 – regardless of the standard character set.

## Possible Solutions to Date

For protecting an application against such errors, there have been two possibilities so far:

1. Specify the character set when calling all methods that convert strings to bytes and vice versa.
2. Set the default character set via system property "file.encoding".

**The first option leads to a lot of code duplication and is thus messy and error-prone:**

FileWriter fw = new FileWriter("happy-coding.txt", StandardCharsets.UTF\_8);

// ...

FileReader fr = new FileReader("happy-coding.txt", StandardCharsets.UTF\_8);

// ...

Files.readString(Path.of("happy-coding.txt"), StandardCharsets.UTF\_8);

**Specifying the character set parameters also prevents us from using method references, as in the following example,**

Stream<String> encodedParams = ...

Stream<String> decodedParams = encodedParams.map(URLDecoder::decode);

Instead, we would have to write:

Stream<String> encodedParams = ...

Stream<String> decodedParams =

encodedParams.map(s -> URLDecoder.decode(s, StandardCharsets.UTF\_8));

The second possibility (system property "file.encoding") was firstly not officially documented up to and including Java 17 (see system properties documentation).

Secondly, as explained above, the character set specified is not used for all API methods. So the variant is also error-prone, as we can show with the example from above:

**public class Jep400Example {**

**public static void main(String[] args) throws IOException {**

**try (FileWriter fw = new FileWriter("happy-coding.txt");**

**BufferedWriter bw = new BufferedWriter(fw)) {**

**bw.write("ハッピーコーディング！");**

**}**

**String text = Files.readString(Path.of("happy-coding.txt"));**

**System.out.println(text);**

**}**

**}**

Let's run the program once with standard encoding US-ASCII:

$ java -Dfile.encoding=US-ASCII Jep400Example.java

**?????????????????????????????????**

The result is garbage because FileWriter takes the default encoding into account, but Files.readString() ignores it and always uses UTF-8. So this variant only works reliably if you use UTF-8 uniformly:

**$ java -Dfile.encoding=UTF-8 Jep400Example.java**

ハッピーコーディング！

**JEP 400 to the Rescue**

With JDK Enhancement Proposal 400, the problems mentioned above will – at least for the most part – be a thing of the past as of Java 18.

The default encoding will always be UTF-8 regardless of the operating system, locale, and language settings.

Also, the system property "file.encoding" will be documented – and we can use it legitimately. However, we should do this with caution. The fact that the Files methods ignore the configured default encoding will not be changed by JEP 400.

According to the documentation, only the values "UTF-8" and "COMPAT" should be used anyway, with UTF-8 providing consistent encoding and COMPAT simulating pre-Java 18 behavior. All other values lead to unspecified behavior.

Quite possibly, "file.encoding" will be deprecated in the future and later removed to eliminate the remaining potential source of errors (methods that respect the default encoding vs. those that do not).

The best way is always to set "-Dfile.encoding" to UTF-8 or omit it altogether.

The current default encoding can be read at runtime via Charset.defaultCharset() or the system property "file.encoding". Since Java 17, the system property "native.encoding" can be used to read the encoding, which – before Java 18 – would be the default encoding if none is specified:

System.out.println("Default charset : " + Charset.defaultCharset());

System.out.println("file.encoding : " + System.getProperty("file.encoding"));

System.out.println("native.encoding : " + System.getProperty("native.encoding"));

Without specifying -Dfile.encoding, the program prints the following on Linux and macOS with Java 17 and Java 18,

**Default charset : UTF-8**

**file.encoding : UTF-8**

**native.encoding : UTF-8**

On Windows and Java 17, the output is as follows,

**Default charset : windows-1252**

**file.encoding : Cp1252**

**native.encoding : Cp1252**

And on Windows and Java 18:

**Default charset : UTF-8**

**file.encoding : UTF-8**

**native.encoding : Cp1252**

So the native encoding on Windows remains the same, but the default encoding changes to UTF-8 according to this JEP. The Previous "Default" Character Set

If we run the little program from above on Linux or macOS and Java 17 with the -Dfile.encoding=default parameter, we get the following output,

**Default charset : US-ASCII**

**file.encoding : default**

**native.encoding : UTF-8**

This is because the name "default" was previously recognized as an alias for the encoding "US-ASCII".

In Java 18, this is changed: "default" is no longer recognized; the output looks like this,

**Default charset : UTF-8**

**file.encoding : default**

**native.encoding : UTF-8**

The system property "file.encoding" is still "default" – but at this point, we would also see any other invalid input. The default character set for an invalid "file.encoding" input is always UTF-8 as of Java 18 or corresponds to the native encoding up to Java 17.

**Charset.forName() Taking Fallback Default Value**

Not part of the above JEP and not defined in any other JEP is the new method Charset.forName(String charsetName, Charset fallback). This method returns the specified fallback value instead of throwing an IllegalCharsetNameException or an UnsupportedCharsetException if the character set name is unknown or the character set is not supported.

public class \_01\_CharSet\_Default\_UTF\_Demo1 {

public static void main(String[] args) throws IOException {

System.***out***.println("Default CharSet ==> " + Charset.*defaultCharset*());

System.***out***.println("file.encoding ==> " + System.*getProperty*("file.encoding"));

System.***out***.println("native.encoding ==> " + System.*getProperty*("native.encoding"));

try (FileWriter fw = new FileWriter("happy-learning.txt"); BufferedWriter bw = new BufferedWriter(fw)) {

bw.write("ハッピーコーディング！");

} catch (IOException ex) {

ex.printStackTrace();

}

String text = Files.*readString*(Path.*of*("happy-learning.txt"));

System.***out***.println(text);

}

}

# JEP 420 - Pattern Matching for Switch

This JEP is the second preview of pattern matching for the switch, with the following enhancements since the first preview:

## Dominance checking of the same type.

Review the below switch pattern matching, every value that matches the String s also matches the CharSequence cs, which makes String s unreadable and will cause a compile-time error.

**static void error (Object o){**

**switch (o) {**

**case CharSequence cs -> System.out.println("A sequence of length " + cs.length());**

**case String s -> // Error - pattern is dominated by previous pattern**

**System.out.println("A string: " + s);**

**default -> {**

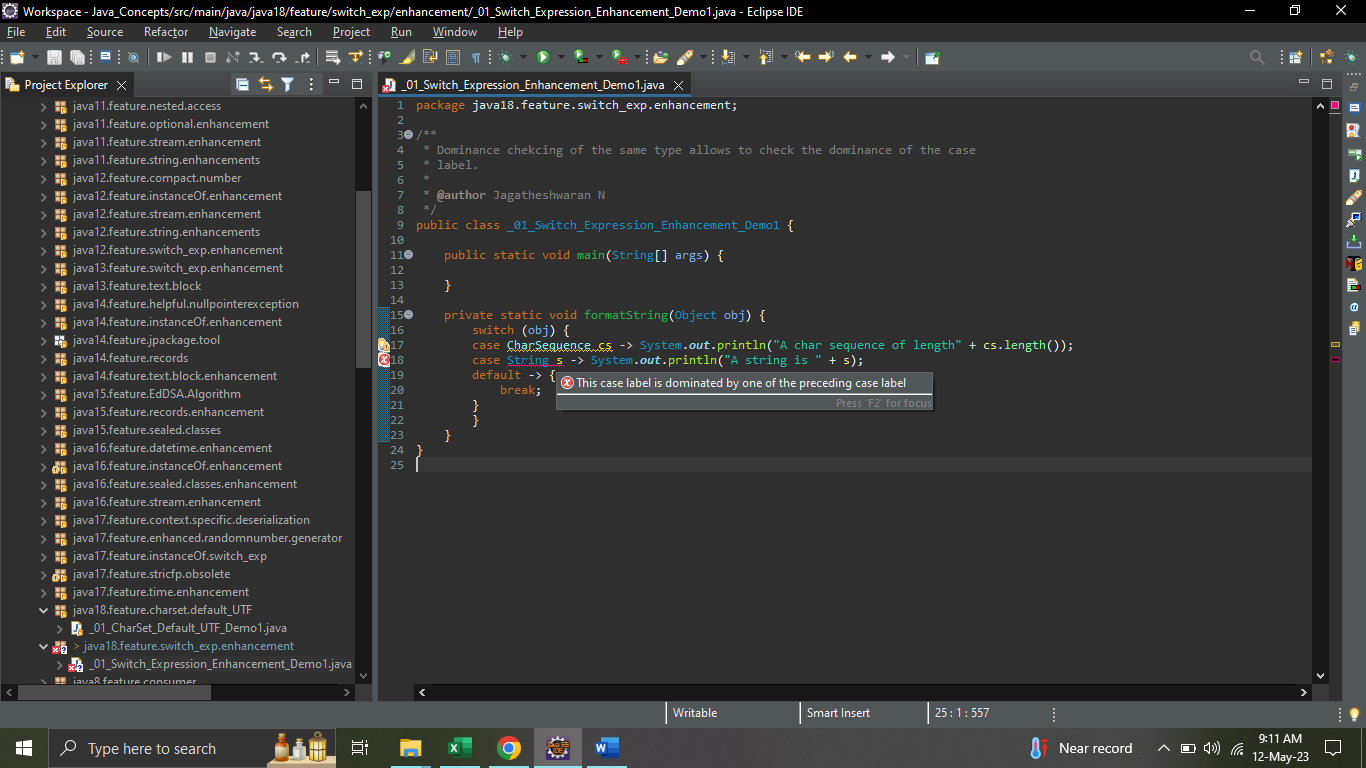
**break;**

**}**

**}**

**}**

If we run the above code with Java 18, it will cause a compile-time error.



## Exhaustiveness of switch expressions and statements

The switch expression requires all possible values to be handled in the switch block, else prompts a compile-time error.

**Review the below code:**

static int coverage(Object o) {

return switch (o) { // Error - not exhaustive

case String s -> s.length();

case Integer i -> i;

};

}

If we run the above code with Java 18, it will cause a compile-time error.

**java: the switch expression does not cover all possible input values**

The below code is fine because the default will handle all the possible types.

static int coverage(Object o) {

return switch (o) {

case String s -> s.length();

case Integer i -> i;

default -> 0;

};

}

public class \_02\_Switch\_Expression\_Enhancement\_Demo2 {

public static void main(String[] args) {

System.***out***.println(*checkExp*("John"));

}

*@SuppressWarnings*("preview")

private static int checkExp(Object obj) {

// Exception in thread "main" java.lang.Error: Unresolved compilation problem:

// A switch expression should have a default case

// return switch (obj) {

// case String s -> s.length();

// case Integer i -> i;

// };

return switch (obj) {

case String s -> s.length();

case Integer i -> i;

default -> 0;

};

}

}

# JEP 408 - Simple Web Server

Almost all modern programming languages allow starting up a rudimentary HTTP server to, for example, quickly test some web functionality.

Through **JDK Enhancement Proposal 408**, Java also offers this possibility as of version 18.

The easiest way to start the provided webserver is the jwebserver command. It starts the server on localhost:8000 and provides a file browser for the current directory:

**$ jwebserver**

**Binding to loopback by default. For all interfaces use "-b 0.0.0.0" or "-b ::".**

**Serving /home/sven and subdirectories on 127.0.0.1 port 8000**

**URL http://127.0.0.1:8000/**

As shown, you can use the -b parameter to specify the IP address on which the server should listen. With -p, you can change the port and with -d the directory the server should serve. With -o, you can configure the log output. For example:

**$ jwebserver -b 127.0.0.100 -p 4444 -d /tmp -o verbose**

**Serving /tmp and subdirectories on 127.0.0.100 port 4444**

**URL http://127.0.0.100:4444/**

You get a list of options with explanations with jwebserver -h.

**Web Server Features**

The web server is very rudimentary and has the following limitations,

The only supported protocol is HTTP/1.1.

HTTPS is not provided.

Only the HTTP GET and HEAD methods are allowed.

Java API: SimpleFileServer

jwebserver is not a standalone tool, but just a wrapper that calls:

java -m jdk.httpserver

This command calls the main() method of the sun.net.httpserver.simpleserver.Main class of the jdk.httpserver module, which, in turn, calls SimpleFileServerImpl.start(…). This starter evaluates the command line parameters and finally creates the server via SimpleFileServer.createFileServer(…).

With this method, you can also start a server via Java code:

**HttpServer server =**

**SimpleFileServer.createFileServer(**

**new InetSocketAddress(8080), Path.of("\tmp"), OutputLevel.INFO);**

**server.start();**

Using the Java API, you can extend the web server. You can, for example, make specific directories of the file system accessible via different HTTP paths, and you can extend the server with your own handlers for certain paths and HTTP methods (e.g., PUT).

# JEP 418 - Internet-Address Resolution SPI

By default, the java.net.InetAddress API uses the operating system’s built-in resolver to resolve host names to Internet Protocol (IP) addresses.

**InetAddress ip = InetAddress.getByName("google.com");**

This JEP redesign java.net.InetAddress API to use service loader to find the resolver instead of using the operating system’s built-in resolver.

**InetAddress.java**

**private static InetAddressResolver loadResolver() {**

**return ServiceLoader.load(InetAddressResolverProvider.class)**

**.findFirst()**

**.map(nsp -> nsp.get(builtinConfiguration()))**

**.orElse(BUILTIN\_RESOLVER);**

**}**

# Reference

<https://www.happycoders.eu/java/java-18-features/>

<https://mkyong.com/java/what-is-new-in-java-18/>